Parallel Copy Motion

Florent Bouchez, Quentin Colombet, Alain Darte, Christophe Guillon, Fabrice Rastello

Title omitted for double-blind reasons

Florent Bouchez Quentin Colombet Alain Darte
ENS-Lyon
firstname.lastname@ens-lyon.fr

Christophe Guillon
STMicroelectronics
firstname.lastname@st.com

Fabrice Rastello
ENS-Lyon
firstname.lastname@ens-lyon.fr

Abstract

Recent results on the static single assignment (SSA) form open promising directions for the design of new register allocation heuristics for just-in-time (JIT) compilation. In particular, heuristics based on tree scans with two decoupled phases, one for spilling, one for splitting/coloring/coalescing, seem good candidates for designing memory-friendly, fast, and competitive register allocators. Another class of register allocators, well-suited for JIT compilation, are those based on linear scans. Most of them perform coalescing poorly but also do live-range splitting (mostly on control-flow edges) to avoid spilling. This leads to a large amount of register-to-register copies inside basic blocks but also, implicitly, on critical edges, i.e., edges that flow from a block with several successors to a block with several predecessors.

This paper presents a new back-end optimization that we call parallel copy motion. The technique is to move copy instructions in a register-allocated code from a program point, possibly an edge, to another. In contrast with a classical scheduler that must preserve data dependences, our copy motion also permutes register assignments so that a copy can "traverse" all instructions of a basic block, except those with conflicting register constraints. Thus, parallel copies can be placed either where the scheduling has some empty slots (for multiple-issues architectures), or where fewer copies are necessary because some variables are dead at this point. Moreover, to the cost of some code compensations (namely, the reverse of the copy), a copy can also be moved out from a critical edge. This provides a simple solution to avoid critical-edge splitting, especially useful when the compiler cannot split it, as it is the case for abnormal edges. This compensation technique also enables the scheduling/motion of the copy in the successor or predecessor basic block.

Experiments with the SPECint benchmarks suite and our own benchmark suite show that we can now apply broadly an SSA-based register allocator: all procedures, even with abnormal edges, can be treated. Simple strategies for moving copies from edges and locally inside basic block show significant average improvements (4% for SPECint and 3% for our suite), with no degradation. It let us believe that the approach is promising, and not only for improving coalescing in fast register allocators.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation, Compilers, Optimization

General Terms Algorithms, Languages, Performance, Theory

Keywords Register allocation, Register copies, Critical edge

1. Introduction

In back-end code generators, register coalescing means allocating to the same register the two variables involved in a `move` instruction so that the copy becomes useless. The register coalescing problem is the corresponding optimization problem, i.e., how to map variables to registers so as to reduce the cost of the remaining copies. Before quite recently, this issue was not very important because, usually, the codes obtained after optimization did not contain many `move` instructions. Even if they did, register coalescing algorithms, such as the iterated register coalescing (IRC) [9], were good enough to eliminate most of them. Today, the context of just-in-time (JIT) compilation and of static single assignment (SSA) has put the register coalescing problem in the light again and raised new problems.

The time and memory footprint constraints imposed by JIT compilation have led to the design of cheap register allocators, most of them derived from a "linear scan" approach [14, 16, 18, 19]. These algorithms perform a simple traversal of the basic blocks, without building any interference graph, in order to save compilation time and space. To make the technique work, `move` instructions need to be introduced on control-flow edges so that the register allocations made for previous predecessor blocks match. Since these register allocators are designed to be fast, they usually use cheap heuristics that may lead to poor performance. In particular, many `move` instructions can remain, which, in addition, can lead to edge splitting, i.e., the insertion, when the edge is "splittable", of a new basic block where register-to-register copies will be performed.

A similar situation occurs in the design of register allocators based on two decoupled phases. In such allocators, a first phase performs spilling (i.e., `load` and `store` insertions) so that the register pressure (i.e., the maximal number of variables simultaneously live) is less than the number of available registers. Then, a second phase allocates the remaining live ranges of variables to the registers, with no additional spill. For this to be possible, live-range splitting may be necessary, i.e., `move` instructions may need to be introduced so that variables are not constrained to be in the same register during their whole live range. Such a strategy is appealing because the spilling problem and the coloring/coalescing problem can be treated separately, thus possibly with cheaper algorithms. The underlying assumption is that it is preferable to insert (possibly many) `move` instructions if this can avoid inserting `load` and `store` instructions. A key point however is to decide how to split live ranges so that the coloring with no additional spill is feasible. A possible approach is to split aggressively, i.e., to introduce `move` instructions possibly between any two instructions [1]. This creates an enormous amount of new variables, which in turn makes the interference graph very big, and introduces many `move` instructions that should be eliminated. Another possibility is to rely on the live-

range split induced by SSA, thus to introduce `move` instructions at the dominance frontier [4, 5, 12]. Both approaches may introduce `move` instructions – actually, sets of parallel `move` instructions – on control-flow edges, which requires, again, edge splitting.

These two situations illustrate the need for a better way of handling parallel copies: some JIT algorithms perform coalescing poorly, so a fast and better coalescing scheme is needed, and some algorithms (JIT or not) rely on the insertion of basic blocks, i.e., on edge-splitting, which is not always desirable:

- edge splitting adds one more instruction (a `jump`), a problem on highly-executed edges;
- splitting the back-edge of a loop may block the use of a hardware loop accelerator;
- compilers may insert abnormal edges, i.e., control-flow edges that cannot be split (for computed goto extensions, exception support, or region scoping);
- copies inserted on critical edges cannot be scheduled efficiently without additional scheduling heuristics (speculation, compensation), especially on multiple-issues architectures.

The goal of this paper is to propose a general framework for moving around parallel copies in a register-allocated code. Section 2 illustrates the concept of parallel copy motion inside a basic block and out of a control-flow edge. For a critical edge, i.e., an edge from a block with more than one successor to a block with more than one predecessor, moving copies is more complicated, as some compensation on adjacent edges must be performed, then possibly propagated. Section 3 describes more formally our method, which is based on moving permutations of register colors (possibly with compensation). In Section 4, we develop simple heuristics to optimize the placement of moved parallel copies and address our initial problems, i.e., parallel copy motion for better coalescing and to avoid edge splitting. Section 5 shows the results of our experiments on SPECint benchmark suites. We show in particular that it is better *not* to split edges everywhere, but to move copies instead. We conclude in Section 6.

2. Parallel copy motion and compensation

2.1 Parallel copies

Parallel copies are virtual instructions that perform multiple `move` instructions *at the same time*. The moves represent the flow of values that must be performed by the parallel copy. The parallel semantics is fundamental, since performing moves in a sequential way with no care may cause a value to be erased before being copied to its proper destination, variable or register.

As recalled in Section 1, register-to-register parallel copies come from live-range splitting during register allocation. In particular, in most extensions of the linear-scan register allocator, the assignment of a variable between two consecutive basic blocks might be different, which leads, implicitly, to a register-to-register parallel copy *on the edge* between the two basic blocks. Fig. 1a illustrates such a case: the registers assigned to a and b in basic block B_d are swapped compared to the assignment in B_s, hence, the values contained in R_1 and R_2 must be swapped on the edge from B_s to B_d. On the contrary, variable c is assigned to R_3 on the two basic blocks so the value of R_3 should remain there. Similarly, when performing SSA-based register allocation, the semantics of ϕ-functions is similar to parallel copies on incoming edges. The removal of ϕ-functions leads to the introduction of parallel copies on edges. Fig. 1b shows an example where R_1 and R_2 must be swapped on the left edge, because the left arguments of ϕ-functions are in different registers than the variables defined.

In these contexts, a parallel copy means that values must be transferred between registers from one program point to another. For this reason, it is handy to represent, in the parallel copy, the registers that keep their value in place. In other words, we enforce a parallel copy to represent the liveness because all "interesting" values, i.e., those of live variables, are referenced in the parallel copy. A parallel copy can be represented as a graph in which live registers are nodes and directed edges represent the flow of the values [10, 13, 15]. Self-edges are necessary to represent unmodified but live registers. In short, R_i is in the live-in (resp. live-out) set of the parallel copy iff there is an edge leaving (resp. entering) the node R_i in the graph representation. For simplicity, we consider that any register in the graph representation of the parallel copy has at most one entering edge. Otherwise, this would mean that the two source registers carry the same value. In such case, we should modify the code so that it uses only one of the registers at this point.

Finally, we also consider that there is at most one edge leaving a register. We call such a parallel copy, a *reversible parallel copy*. The advantage of this restriction will appear clearly in the next section. Actually, when going out of SSA, it is possible that the removing of ϕ-functions creates "duplications" in parallel copies: the value of one register gets copied to two or more registers. This happens for instance if, at the beginning of a basic block, the same variable is used twice as argument, as in $[b \leftarrow \phi(a,\ldots); c \leftarrow \phi(a,\ldots)]$, or if two arguments have been coalesced and renamed into one variable. In practice, the duplications can be extracted from the parallel copies and placed in the predecessor basic block. But this task may lead to additional spilling and we choose for clarity not to treat this case here. None of the existing linear-scan register allocators would lead to parallel copies with duplications on edges. For people concerned by SSA-based register allocators, the aforementioned situation could be avoided beforehand by first adding, before the allocation, the duplicating copies in the predecessor basic blocks. This is less constraining than enforcing SSA to be conventional static single assignment (CSSA) [17], but CSSA would do the job [13].

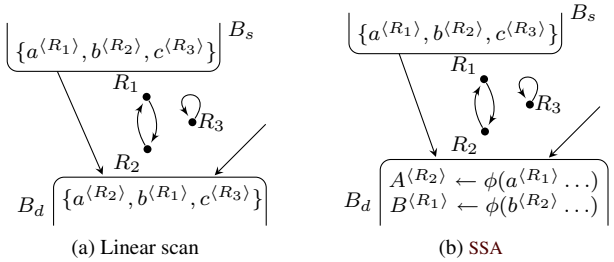
With these constraints in mind, a reversible parallel copy can be defined from its *live_in* and *live_out* sets as follows:

Definition 1. A reversible parallel copy is a one-to-one mapping from its *live_in* set to its *live_out* set. We use the following notation: $/\!\!/c : (v_1,\ldots,v_n) \leftarrow (a_1,\ldots,a_n)$ where $/\!\!/c(a_i) = v_i$ and $/\!\!/c^{-1}(v_i) = a_i$.

In our case, live-sets are subsets of the register set. Note that *live_in* and *live_out* are not necessarily disjoint. In terms of graph representation, a parallel copy is a set of disjoint subgraphs, where each subgraph is either a chain or a simple cycle. For $R_i \notin$ *live_in*, we abusively write $/\!\!/c(R_i) = \perp$ and, for $R_i \notin$ *live_out*, $/\!\!/c^{-1}(R_i) = \perp$.

2.2 Moving a parallel copy out of an edge

Critical edges are edges of the control-flow graph (CFG) from a basic block with multiple successors to a basic block with multiple predecessors (e.g., the bold edge of Fig. 2a). It is obvious to move a parallel copy out of a non-critical edge. It can indeed be placed at the bottom (resp. top) of the source (resp. destination) block, if this block has only one successor (resp. predecessor). This is not directly possible for critical edges, as the parallel copy would then



(a) Linear scan (b) SSA

Figure 1: Parallel copies on edges.

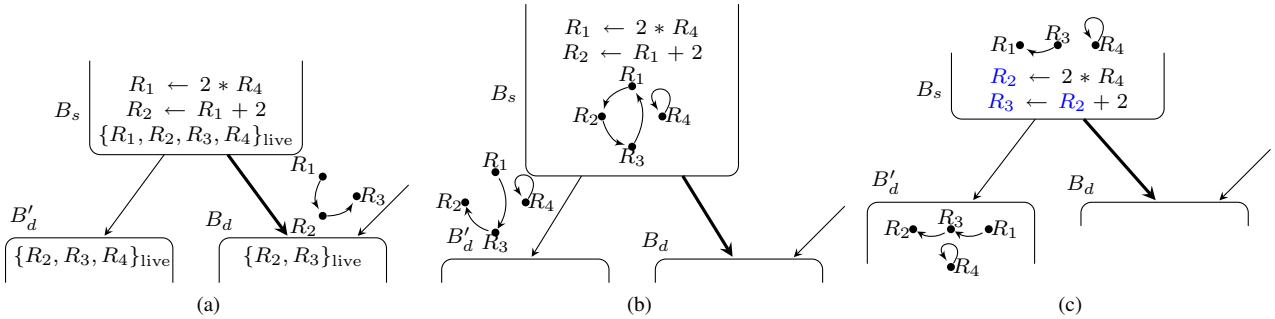


Figure 2: On a critical edge (a), parallel copies can be moved if compensated; (b) the parallel copy is augmented to include the liveness of the top basic block, and is compensated on the other leaving edge; (c) the permutation is moved higher in the basic block and its size may shrink (here it does), the compensation code is put at the beginning of the left basic block.

be executed on other undesired paths. However, it is possible to *compensate* the effect of a reversible parallel copy on other edges.

This is similar to the idea introduced by [7] for trace scheduling, later called "compensation code", but it concerns general code and deals only with duplicating the code when moving instructions above a join point or below a split point. According to [8], it has been suggested that code could be inserted to undo any effects on "off-trace paths", but it is not done in practice because, even if it would be possible for simple register operations, it is too complex for general operations. We present in this section a way to "undo" the effects caused by reversible parallel copies.

When trying to move a reversible parallel copy away from a critical edge $E : B_s \rightarrow B_d$, there are two possibilities: either move it *down*, i.e., to the top of B_d, or move it *up*, i.e., to the top of B_s, as done in Fig. 2b. As illustrated by this example, when moving a parallel copy up, it might be expanded to reflect the change of liveness between the critical edge and the end of the predecessor basic block. In our example, the reversible parallel copy grows with a self edge on R_4 and an edge from R_3 to save its value in R_1. Indeed otherwise, the transfer from R_2 to R_3 would overwrite the value of a live variable, stored in R_3 and needed in B'_d.

Once the parallel copy has been moved up, its effect should be compensated on the other outgoing edges. The compensation is roughly the reverse of the parallel copy. This explains why we restricted initial parallel copies on edges to be reversible. In Fig. 2b, the values of R_2 and R_3, which are alive on B'_d must be restored.

This example shows that a reversible parallel copy can be moved out of a critical edge, at the price of some compensation code. Then, the critical edge does not have to be split anymore, and the copies can now be scheduled with the other instructions of the block B_s. The same is true for the compensation code on B'_d. The precise mechanism to perform this transformation correctly is explained in the next section using the notion of permutation motion.

2.3 Parallel copy motion inside basic blocks

It is also possible to move a parallel copy inside a basic block. The trick is to consider the parallel copy as a reassignment function and not as a general instruction. This is of course possible only by reassigning operands of "traversed" instructions. Fig. 2c shows an example where, after having moved a parallel copy up from an edge, the copy is further moved inside the basic block. The operands in blue have been reassigned accordingly. Here, the resulting parallel copy is smaller after being moved up because R_1 and R_2 are not alive before, thus their values do not need to be transferred.

The details for performing this transformation will use the *permutation motion* and *region recoloring* concepts. As illustrated by this example, one of the benefits of moving a reversible parallel copy inside a basic block is that its size may shrink down because the liveness changes. Another potential advantage of this technique, not developed in this paper, is the ability to place part of the reversible parallel copies on empty slots of a scheduling.

One restriction to the motion inside basic blocks concerns registers constraints. Indeed, some instruction operands cannot be reassigned, for example for function calls. So, unless $/\!\!/c(R_i) = R_i$ for all constraints of an instruction I, $/\!\!/c$ cannot be moved beyond I as it is. Still, it does not mean that we are blocked. It is in fact possible to decompose $/\!\!/c$ into $/\!\!/c' \circ /\!\!/c_{id}$ where $/\!\!/c_{id}$ is the identity for all register constraints of I. Then, $/\!\!/c'$ stays on its side of I while $/\!\!/c_{id}$ can be moved further. We will not develop this any further in this article.

3. Permutation motion & region recoloring

To take liveness into account when moving reversible parallel copies, we propose a solution based on *permutations*.

Definition 2. A *permutation* is a one-to-one mapping from the whole set of registers to the whole set of registers.

As seen previously, moving a reversible parallel copy should be done carefully because of liveness. A permutation is a transfer function that does not have to cope with liveness. Because of that, it is much easier to move it. The idea here is to extend a reversible parallel copy into a permutation (we call it expansion), then to move the permutation, and finally to transform back the permutation to a reversible parallel copy (we call it projection).

3.1 Reversible parallel copies to/from permutations

Let Live be the set of registers that contain a live value at program point p. Placing a permutation π at p has the effect of moving each register R_i into $\pi(R_i)$. However, only registers in Live need to be considered as other registers contain useless values. We can then define a (reversible) parallel copy $proj(\pi)$, the projection of π, as the restriction of π to the registers in Live. In other words, the *live_in* set of $proj(\pi)$ is Live, its *live_out* set is the image of Live by π, and $\forall R_i \in$ Live, $proj(\pi)(R_i) = \pi(R_i)$. In the graph representation, all edges leaving registers that do not contain any live value can be safely removed. All remaining edges move data of a live variable and hence must remain in the projected permutation.

Expanding a reversible parallel copy amounts to find a permutation whose projection is the initial reversible parallel copy. First, the *live_in* set must be augmented to be the whole set of registers. Second, since a permutation contains only cycles, the chains of the reversible parallel copy must be closed to form permutation cycles. Of course, there are more than one way to expand a parallel copy.

We propose a pseudo-code, Function Expand below. For every register that still has no predecessor (Line 3), i.e., every beginning of a chain, the loop Line 5 finds the register at the end of the chain. It then connects this register to the first one so as to form a cycle. Free registers are made cycles of length one (self-loop) by this process. This way, π is the identity for as many registers as possible. Another possibility is to turn all chains into a unique cycle so that it can be "sequentialized" [3] with a minimum number of swaps.

```
Function Expand(∥c)
    Data: Parallel copy ∥c.
    Output: Permutation π, an expansion of ∥c.
1   π = ∥c ;                                   /* Make π a copy of ∥c. */
2   foreach R_i ∈ Registers do
3       if π^{-1}(R_i) = ⊥ then
4           current ← R_i ;
5           while π(current) ≠ ⊥ do current ← π(current);
6           π(current) ← R_i ;        /* Close the chain to form a cycle */
7   return π;
```

3.2 Region recoloring

We can define the permutation motion mechanism in a more formal way by noticing that, for any region of the program, i.e., any set of instructions, it is possible to add a permutation π at every entry point of the region, to add its inverse π^{-1} at every exit point of the region, and to reassign every operand in the region according to π: textually replace inside the region every occurrence of R_i by $\pi(R_i)$. However, there are still limitations to this, as for the motion of parallel copies in a basic block described earlier: some instructions have register constraints, e.g., arguments of a `call`, that cannot be recolored. So, unless $\pi(R_i) = R_i$ for all such constraints, these instructions cannot be part of such a region.

We call this alternative view of permutation motion *region recoloring*, since the variables of the regions get reassigned to different registers. Using this formalism, it is easy to understand how to move a permutation in a basic block, and more generally how the whole parallel copy motion works. On Fig. 3, the reversible parallel copy $\parallel c$ will be moved up into basic block B_s by recoloring the grey region with an expansion π of $\parallel c$: on the right edge, the composition of proj(π) followed by $\parallel c$ simplifies to the identity.

Let us illustrates the process on the example of Fig. 2 with 4 registers R_1 to R_4 and the same region recoloring as in Fig. 3. A possible expansion of the reversible parallel copy $(R_2, R_3) \leftarrow (R_1, R_2)$ is to extend it with $\pi(R_3) = R_1$, i.e., $\pi : (R_2, R_3, R_1, R_4) \leftarrow (R_1, R_2, R_3, R_4)$. The projection of π at the top of B_s is $(R_4) \leftarrow (R_4)$ as the initial live-in of the region $\{R_4\}$ must match the live-in of the reversible parallel copy. The projection of π^{-1} on B'_d is $(R_2, R_3, R_4) \leftarrow (R_3, R_1, R_4)$ (the initial live-out of the region $\{R_2, R_3, R_4\}$ must match the live-out of the reversible parallel copy. Within the region, R_1 is replaced by $\pi(R_1) = R_2$, R_2 is replaced by $\pi(R_2) = R_3$, there is no occurrence of R_3, and R_4 is unchanged.

To move a reversible parallel copy $\parallel c$ out of the edge E from B_s to B_d, let π be an expansion of $\parallel c$. If one wants to move $\parallel c$ up (for instance), let us choose any convenient region with an entry point somewhere inside B_s, and exit points on every edge leaving B_s. First, π is added at the entry and π^{-1} at every exit. The variables of the region in between are reassigned according to π. At the entry point inside B_s, π is projected. On the edge E, the projection of π^{-1} and $\parallel c$ cancel each other and no code remains. On all other edges, π^{-1} is projected. If these edges are not critical, π^{-1} can be projected at the top of their corresponding destination basic blocks, or even deeper in the blocks if wanted. Section 4.1.2 discusses the case where some of these edges are critical.

To conclude, while trying to move directly reversible parallel copies seems awkward and mind twisting, the detour through permutation motion and region recoloring shows that parallel copy motion is, in fact, not a difficult task to perform. The last task is then to *sequentialize* the parallel copies using actual instructions of the target architecture. This is a standard operation, see for example [3]. The only critical case is when the parallel copy permutes all registers, in which case a swap mechanism is needed.

4. Applications

We now detail some applications of parallel copy motion.

4.1 Remove parallel copies from critical edges

The problem with parallel copies on edges is that there is no basic block there. So, in order to actually add code, such an edge must be split and a new basic block must be created to hold the instructions. However, as mentioned in Section 1, there is a folk assumption that splitting edges is a bad idea. The main reasons are both performance reasons (additional jump instruction, prevents the use of hardware loops, interaction with basic block scheduling) and functional reasons (abnormal edges inserted by compilers).

We now show how to optimize the removal of parallel copies out of control-flow edges. We first present, in Section 4.1.1, a heuristic based on a local cost function to decide if an edge should be split or if the parallel copy it contains should be moved. This mechanism can fail if parallel copies are moved out of an unsplittable edge whose neighboring edges are also unsplittable. This situation is addressed in Section 4.1.2 and a simple propagation mechanism along critical edges is proposed.

4.1.1 A local heuristic

The input of the heuristic is a CFG graph with a reversible parallel copy, possibly the identity, on each control-flow edge and at the top and bottom of each basic block. The principle of the heuristic is to deal first with edges that cannot be split, and then to deal with the others in decreasing order of frequency. For each edge in a worklist (initialized with all edges with a parallel copy different than the identity), the heuristic evaluates the impact of parallel copy motion (moving it up, moving it down) by computing a local gain (possibly negative) compared to the solution that keeps the parallel copy on the edge, i.e., compared to edge-splitting. Then, the heuristic chooses the best feasible solution, applies the modifications, and removes the edge from the worklist. When the content of another edge is modified (because the parallel copy was moved and compensated as explained in Section 3), it is added (if not already) in the worklist unless its new parallel copy is the identity. The heuristic continues until the worklist gets empty, i.e., it stops when no reversible parallel copy motion leads to a positive gain. Of course, the possibility of staying on the edge is not feasible for non-splittable edges. Likewise, a moving is not feasible if it produces a parallel copy, different than the identity, on a non-splittable edge. If none of the possibilities is feasible, then the heuristic fails. This case is discussed in Section 4.1.2.

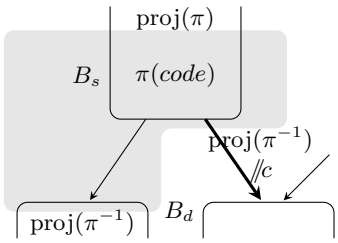


Figure 3: Region recoloring, starting with $\parallel c$ on the critical edge.

To evaluate the gain, the heuristic should simulate the motion and the compensation on neighboring edges using a performance model. To illustrate the heuristic, let us describe a toy model for a very-long instruction word (VLIW) architecture with 4 issues:

- In a block B of frequency W_B, an instruction costs $\widehat{\text{inst}} = \frac{1}{4} \times W_B$.
- The cost of splitting an edge E of frequency W_E is the cost of a jump, $\widehat{\text{inst}}$, plus the branch penalty $1 \times W_E$, thus $\widehat{\text{split}} = \frac{5}{4} \times W_E$.
- The number of copy instructions, $\|/\!/c\|$, necessary to sequentialize a parallel copy $/\!/c$ is the number of (non-self) edges plus the number of cycles in its graph representation.
- The cost of a set of copies is different if placed on an edge (cannot be scheduled with other instructions) $\widehat{/\!/c} = \left\lceil \frac{\|/\!/c\|}{4} \right\rceil \times W_E$ than on a basic block $\widehat{/\!/c} = \frac{\|/\!/c\|}{4} \times W_B$.

Of course this model is far from being perfect, but the effect of further optimizations (e.g., post-pass scheduler), in addition to the approximation made on edge frequency, makes it difficult to model more precisely. What we need is just a model to drive the heuristic in the right direction. Consider as an example the code of Fig. 5(a). If we leave the parallel copy in place, the local cost will be evaluated as $\left\lceil \frac{1}{4} \right\rceil \times W_{(AB,B)} + \frac{5}{4} \times W_{(AB,B)}$. If we move it down, the cost will be evaluated as $\frac{1}{4} \times W_B + \left\lceil \frac{1}{4} \right\rceil \times W_{(BC,B)} + \frac{5}{4} \times W_{(BC,B)}$. If we move it up, the cost will be evaluated as $\frac{2}{4} \times W_{AB} + \left\lceil \frac{1}{4} \right\rceil \times W_{(AB,A)} + \frac{5}{4} \times W_{(AB,A)}$. Suppose that moving it down leads to a positive gain. At this point, there should be $(R_1) \leftarrow (R_2)$ on the edge (BC, B) and $(R_2) \leftarrow (R_1)$ at the beginning of basic block B. The content of (BC, B) is modified with a non-trivial parallel copy, so (BC, B) is added to the worklist.

The heuristic itself is not local, as copies can move, progressively, further than to neighboring edges. But the decision to move down, to move up, or to split the edge, is made by a local computation of gain (see the pseudo code of Function `Local-Heuristic`).

This heuristic is illustrated in Fig. 4b, assuming than R_2 and R_4 are not live beyond the control-flow edges. In this example, the local heuristic considers the parallel copy on the critical edge first. It computes the gain of keeping the parallel copy on the edge (0). It computes the gain of moving the parallel copy down. This produces a compensation on the edge on the right with two copies and two other copies in the destination basic block. For this motion, we have a negative gain. It then computes the gain of moving the parallel copy up. This produces a compensation on the edge on the left which, composed with the parallel copy already in place, gives the identity, plus two copies in the source basic block. For this motion, the gain is positive. The best local choice is thus moving the parallel copy up, with the result represented in Fig. 4b. Copies can also be moved further in basic blocks. The result is then presented in Fig. 4c. However, a better solution is even to take into account, for computing the gain, the possibility to move the parallel copies in basic blocks. In this case, the heuristic is not mistaken and it finds the code in Fig. 4d. The heuristic we implemented as this capability.

Function `Local-Heuristic`(*e*, *direction*, *simulate*)

Data: Edge e to be processed, direction of the motion *direction*, Boolean *simulate* (FALSE to apply changes).
Result: *direction*, a valid motion for e, returns also the gain

```
1  /\!/c ← e./\!/c; gain ← 0;
2  if simulate = TRUE then save current state;
   /* Move parallel copy in the related basic block */
3  if direction =↑ then
4      edges ← e.B_s.leaveEdges;   /* edges with compensation */
       /* Live set may grow, expand the parallel copy */
5      /\!/c ← Expand_in(/\!/c, e.B_s.liveOutSet);
6      gain ← e.B_s./\!/c_bottom.cost;   /* add initial cost at end of block */
7      e.B_s./\!/c_bottom ← /\!/c ∘ e.B_s./\!/c_bottom; /* compose to get new copy */
8      gain ← gain − e.B_s./\!/c_bottom.cost;   /* subtract new cost */
9  else if direction =↓ then
10     edges ← e.B_d.enterEdges;   /* edges with compensation */
11     gain ← e.B_d./\!/c_top.cost;   /* add initial cost at start of block */
12     e.B_d./\!/c_top ← e.B_d./\!/c_top ∘ /\!/c;   /* compose to get new copy */
13     gain ← gain − e.B_d./\!/c_top.cost;   /* subtract new cost */
14 else
       /* We want to split e */
15     if simulate = FALSE and e.isSpittable then e.split = TRUE;
16     return e.isSpittable, gain;
17 /\!/c_inv ← /\!/c^-1;
18 foreach e_i ∈ edges do
       /* For e, its composition with /\!/c_tmp will produce identity. */
19     /\!/c_tmp ← /\!/c_inv;
20     gain ← gain + e_i./\!/c.cost;   /* adds initial cost */
       /* Apply compensation on the edge. */
21     if direction =↑ then
           /* Compensation moves down */
           /* live set may shrink, project on live variables */
22         /\!/c_tmp ← proj(/\!/c_tmp, e_i./\!/c.liveInSet);
23         e_i./\!/c ← e_i./\!/c ∘ /\!/c_tmp;   /* compose to get new copy */
24     else
           /* compensation moves up */
25         e_i./\!/c ← /\!/c_tmp ∘ e_i./\!/c;   /* compose to get new copy */
26     gain ← gain − e_i./\!/c.cost;   /* subtract new cost */
27 if simulate = TRUE then restore current state;
28 return TRUE, gain;
```

4.1.2 Parallel copy motion might be stuck

Whenever two critical edges have the same source (or same destination) basic block, this poses a problem as, obviously, they cannot both move their parallel copy on this basic block since compensation would have to take place on the other edge. Hopefully, it may be possible to move the parallel copies on the basic blocks attached at the other extremities of these edges. Hence, one should not move parallel copies blindly, locally, without taking other edges into account. One should have a look at other "connected" critical edges, which we define as *siblings*. For instance, the two edges leaving AB in Fig. 5 are siblings at the top and the two edges entering B are siblings at the bottom. This situation can also be a problem for the local heuristic of Section 4.1.1. Suppose that, in Fig. 5(a), the edges $AB \to A$, $AB \to B$, and $BC \to B$ are marked as unsplittable. If the first considered edge is the bold edge (from AB to B), then the heuristic fails as it cannot split the edge, and it cannot move the copy neither up, as it would need a compensation on the unsplittable edge $AB \to A$, nor down, as it would need a compensation on the unsplittable edge $BC \to B$. The solution is to have a more global view of chains and cycles of siblings, as we now explain.

We say that a critical edge is weak at bottom (resp. top) if it has no sibling (critical) edge at bottom (resp. top), which means that its parallel copy can be pushed down (resp. up) safely: a local compensation on the other edges will be possible, either by splitting it, or by moving this compensation up (resp. down) to another basic block with no further propagation. The concept of weakness is actually recursive, it can be propagated: if E is a critical edge with a sibling at top E' that is weak (at bottom), then E is weak at top since it is possible to move its parallel copy on its source block, then to add compensation code on E', i.e., compose the compensation code with the existing parallel copy on E', then move down the resulting parallel copy from E' since it is weak at bottom. Therefore, to handle correctly critical edges, either we split them, or we propagate along sibling edges until we reach a non-critical edge. Since we are moving permutations, this is an easy task.

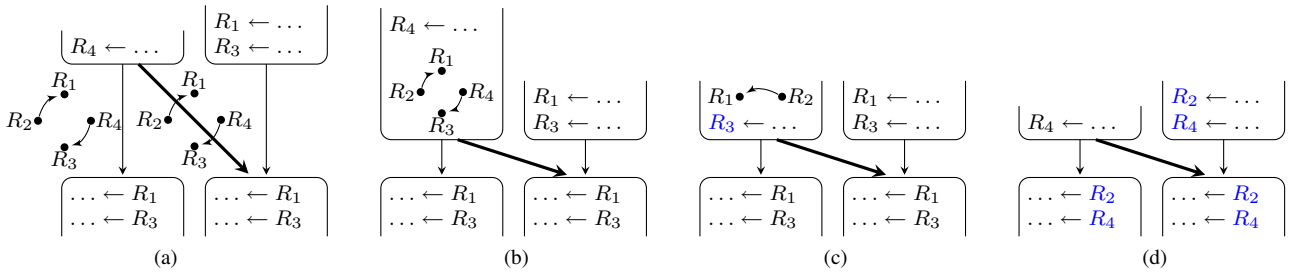
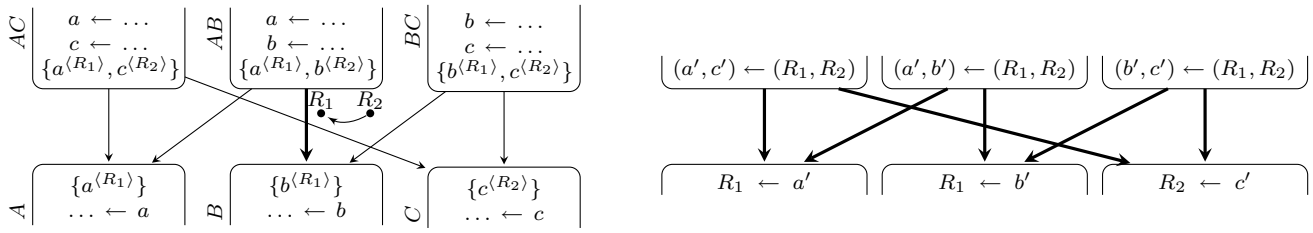


Figure 4: Local heuristic and basic block motion. (a) Initial code, 4 moves; (b) Local heuristic, 2 moves; (c) Local heuristic followed by parallel copy motion in basic block, 1 move; (d) All together, no move.



(a) Local parallel copy motion

(b) Ultimate solution with Chaitin

Figure 5: Complex multiplexing region. (a) The local heuristic can be stuck; (b) an ultimate solution involves Chaitin-like graph coloring.

By a propagation mechanism along the CFG, we can identify all weak edges, those weak at top, those weak at bottom, those weak at both ends. This can be done by propagating from trivially weak edge, following edges alternatively in the control-flow direction or in the inverse direction. Each critical edge encountered along the way is weak at top if the traversal finds it from a sibling at bottom, and weak at bottom if the traversal finds it from a sibling at top. If all edges are finally marked as weak, it is always possible to move all parallel copies out of the control-flow edges. We just have to mark all edges as unsplittable and to consider edges in the right order (for example, by a recursive traversal). Similarly, in the local heuristic, edges should be considered either in the right order, or in a recursive setting. For example, consider again Fig. 5(a), assuming that only the edges $AB \to A$, $AB \to B$, and $BC \to B$ are unsplittable. The local heuristic can push recursively the parallel copy out of $AB \to B$, either down (resp. up), providing that it is pushed later out of $BC \to B$ (resp. $AB \to A$). Or it can proceed edges in order, non recursively, considering the chain $A \to AB \to B \to BC$ either from left or from right.

A last case remains, when there exists a chain of unsplittable sibling edges which, in fact, forms a cycle. This is the case in Fig. 5 if all critical edges are considered as unsplittable. Then, it is not possible to address the problem by propagation, a more global view is needed. We need to consider the whole region formed by the cycle of unsplittable sibling edges and view the problem as a standard graph coloring problem, with coalescing, as in Chaitin-like register allocators. In general, there can be no solution such that all parallel copies are moved out of unsplittable edges. We point out that, in all our benchmarks, we never encountered such a case requiring a global graph coloring approach. Also, for our toy example, a solution exists, as depicted in Fig. 5(b).

4.2 Slim down of parallel copies in a basic block

As mentioned earlier, moving parallel copies out of control-flow edges is not the best we can do. We can still use the parallel copy motion mechanism to move the parallel copy further in the block, either up if it comes from an outgoing edge of the block, or down if it comes from an incoming edge. In a fully-scheduled code, one could look for an empty slot to hide the parallel copy. But even without knowing the schedule, the parallel copy motion can be interesting. Indeed, depending where the parallel copy is placed, the number of moves it implies may vary as the parallel copy is projected on the live variables. For example, the extreme situation is when no variables are live at some program point: placing the parallel copy there means simply recoloring the whole region below (if the copy is moved up), with no move: the parallel copy vanishes. Another side effect is that some remaining moves in the code (with no duplication) can also be absorbed along the way.

We developed a heuristic for parallel copy motion within a basic block. Function `Motion-up-from-bottom` gives the pseudo-code for the motion up (direction ↑) from basic block bottom. The input of the heuristic is a basic block with a reversible parallel copy on its top and its bottom. These parallel copies represent liveness information that may have been composed with the local heuristic of Section 4.1.1. We proceed in two phases. First, we simulate the motion of the parallel copy in the basic block and we record where the parallel copy is the cheapest. In a second time, we do the motion to the previously-selected position. For both the simulation and the motion, we proceed instruction after instruction, updating the cheapest position on the fly. If we cannot traverse an instruction due to coloring constraints, we stop the process (although we could split the parallel copy, as explained in Section 2.3). Moving a parallel copy down in a basic block is similar to the pseudo code of Function `Motion-up-from-bottom`. The only subtlety is to mark last uses, i.e., uses of variables that are not live-out of the instruction.

```
Function Motion-up-from-bottom(block, simulate, position)
  Data: Basic block block where motion is done, Boolean simulate
        (FALSE to apply changes), position where to stop the motion if
        simulate = FALSE.
  Result: Minimum cost after motion, position where cost is minimum.
1 minPosition ← block.bottom;
2 minCost ← block.//c_bottom.cost ;            /* Sequentilization cost */
3 //c_current ← block.//c_bottom;
4 foreach op ∈ block's operations in reverse order do
      /* Current position is both "after op" and "before op.next"   */
5     if simulate = FALSE and current position = position then
6         exit loop;
7     if //c_current can traverse op then
8         foreach result in op's results do
9             dest ← //c_current(result);
10            //c_current(result) ← ⊥ ;   /* result is dead in //c_current */
11            if simulate = FALSE then result ← dest;
          /* Expand liveness if op have last uses colors.               */
12        Expand_in(//c_current, op.liveInSet);
13        if simulate = FALSE then
14            foreach argument in op's arguments do
15                dest ← //c_current(argument);
16                argument ← dest;
17        if //c_current.cost < minCost then
18            minPosition ←before op;
19            minCost ← //c_current.cost;
20    else exit loop;                 /* Happens only when simulating. */
21 if simulate = FALSE then
22    Sequentialize(position, //c_current);
      /* Reset block's parallel copy with the identity on live out set */
23    block.//c_bottom ← Id(block.liveOutSet);
24 return minCost ∗ block.frequency, minPosition;
```

Fig. 2c illustrates this process. After the local heuristic, the parallel copy is moved further up in the basic block. One copy remains before the definitions of R_2 and R_3. In this example, the parallel copy motion is performed after the decision made to move copies out of edges. But, as mentioned earlier, we can integrate the possibility of moving parallel copies inside basic blocks in the cost function given in Section 4.1.1. With no change to the local heuristic, we can achieve better performance. For example, Fig. 4d shows how the new cost function modifies the algorithm decision. Now the parallel copy on the critical edge is moved down, and produces a compensation on the right edge. The resulting parallel copies slim to identity in the related basic block. The same happens for the parallel copy on the left edge. Finally, all copies could be removed, thanks to parallel copy motion, in this example.

5. Experiments

We implemented our parallel copy motion algorithm in the research branch of the code generator of our production compiler. For these experiments, we used it as a static compiler for C code, connected to the OPEN64 compiler. We did not make experiments in the JIT configuration of the compiler as the techniques introduced here have not been implemented in this context yet.

We made our experimentations on the C subset of Spec2000 integer benchmarks and *internal benchmarks* (KERNELS). Our target processor is an embedded VLIW architecture with 4 issues. The *eon* C++ benchmark is not included due to the limited support for C++ in our code generator version. Also the *gap* benchmark is excluded due to a yet unsolved functional problem with our compiler configuration. The KERNELS are a set of computation-intensive kernels like fft, jpeg, and quicksort algorithms, supposed to be representative of embedded media applications as found in firmware code such as audio,video codecs, or image processing.

Benchmark	♯ edges	Benchmark	♯ edges
164.gzip	0	175.vpr	0
176.gcc	117	181.mcf	0
186.crafty	4	197.parser	0
253.perlbmk	55	255.vortex	7
256.bzip2	0	300.twolf	0

Table 1: Number of critical abnormal edges with moves

For this study, we compared the parallel copy motion algorithm against a split-everywhere strategy for critical edges. Both are run after the same biased register coloring heuristic where color selection is biased toward the elimination of copies. We evaluated parallel copy motion algorithm in three modes: motion on edges alone, motion on edges followed by motion inside basic blocks, and motion on edges with both our cost function and motion inside basic blocks. In this section, *edge motion* denotes the heuristic for motion on edges, *block motion* is the motion inside basic blocks, and *all* is the motion on edges using both cost function and motion inside block. When it is not specified otherwise, edge motion is done without block motion. The split-everywhere strategy only splits critical edges when some move operation remain. Other edges are not split as their parallel copies can always be moved, with no compensation, to their source basic block or to their destination basic block.

We show different kind of results, either based on the cost model with static or profile-based basic block frequency estimations, or based on an actual simulation with accurate cycle count. First, at the end of the compilation process, we measured the number and weight of moves, split edges, branches, etc., using basic block frequency estimations as provided by the compiler. These estimations come from some static heuristics derived from [2] for the KERNELS and from edge profiling for Spec2000. Second, we measured actual performance using a cycle-accurate simulator. The performance were measured on the same data set as for the profiling feedback as we want here to illustrate the isolated improvement of our parallel copy motion technique.

5.1 The impact of edge motion

Abnormal edge splitting By using the edge motion algorithm, we were able to get rid of forbidden abnormal edge splitting, for some of the Spec2000 applications, when using our decoupled register allocation algorithm. This experiment is shown on Spec2000 benchmarks only as the KERNELS do not have any abnormal critical edge.

We found 183 critical abnormal edges with remaining move in Spec2000 as reported in Table 1. These edges are present in 4 different applications: gcc, crafty, perlbmk, and vortex. Given the coloring produced by the register allocation heuristic, the compilation of these 4 applications could not be completed without parallel copy motion. Our edge motion algorithm gets rid of all parallel copies on these abnormal edges. Thus on the C subset of the Spec2000, this simple strategy is sufficient to complete the compilation. In particular, this means that butterfly-like patterns (such as Fig. 5) with abnormal edges do not occur, at least in these benchmarks.

Amount of split edge We also measured the number of critical edges that are not split when using our cost model based heuristics, i.e., for which it was preferable to move the parallel copy, according to the model. This shows, as one may expect, that the best insertion point for copies is not always on the edge.

We reduced by a factor of 2 the static number of split edges and using a weighted (by edge frequency) count, we reduced it

Benchmark	Split		Edge motion	
	Number	Weighted	Number	Weighted
164.gzip	1	1	2.2	180.37
175.vpr	1	1	2.31	9.6
176.gcc	1	1	2.67	1.41
181.mcf	1	1	2.45	89.54
186.crafty	1	1	1.92	10.5
197.parser	1	1	2.51	13.95
253.perlbmk	1	1	1.72	16.47
255.vortex	1	1	1.46	2.35
256.bzip2	1	1	2.55	19.38
300.twolf	1	1	2	1.41
G.Mean (10)	1	1	2.14	11.3

Table 2: Reduction factor of critical edges with moves

Benchmark	Split	Edge motion		All
		w/o bl motion	w/ bl motion	
164.gzip	1	1.04	1.03	1.04
175.vpr	1	1.01	1.01	1.02
181.mcf	1	1.04	1.07	1.07
197.parser	1	1.03	1.05	1.05
256.bzip2	1	1.03	1.03	1.03
300.twolf	1	1	1.01	1
G.Mean (6)	1	1.03	1.03	1.04

Table 3: Normalized performance in cycles for Spec2000

by factor of 11. This is expected because our model accounts for the additional branch inserted and for the low resource usage on multiple-issues architectures when an edge is split. In particular, it reflects the fact that a small sequence of operations, as generated by parallel copies, is more costly in a dedicated basic block than on a basic block where it may be scheduled with other operations. Table 2 presents the normalized number and the weight of critical edges which still carry moves at the end of the compilation process.

Performance impact We evaluated the actual performance improvement of our method for the insertion of parallel copies when compiling at aggressive optimization level in our static compiler toolchain. The evaluation was done on the two sets of benchmarks previously presented. Note that we did not give the result for the four Spec2000 benchmarks that cannot be compiled with the split-everywhere strategy on critical edges.

For the Spec2000 with our simple local heuristic, we got an average speedup of 2% with no loss (see Table 3, column edge motion w/o block motion). Two benchmark (gzip, mfc) are improved by up to 4% with this simple heuristic. These encouraging performance results confirm that a split-everywhere strategy not only fails in the case of abnormal edge, but is also inefficient compared to an heuristic based on a cost model to decide if edge splitting is profitable.

Looking at the KERNELS, we also got an average speedup of 2% and no loss (see Table 4, column edge motion w/o block motion). Over the 50 benchmarks, 31 are actually improved. Over these 31 improved benchmarks, seven show a performance speedup of at least 5%. Note that for these tests, we do not use profiling-feedback information, thus even with frequencies estimation, we achieved good results, at least on computation-intensive benchmarks.

5.2 The impact of basic block motion

Weight of moves In order to evaluate the impact of parallel copy motion inside basic blocks, we compared the weight of move operations with the edge motion heuristic and with the edge motion heuristic followed by the basic block motion heuristic.

Benchmark	Split	Edge motion		All
		w/o bl motion	w/ bl motion	
BDTI.bitupck	1	1	1	1
BDTI.bkfir	1	1.02	1.02	1.02
BDTI.bkfir-Copt	1	1.04	1.04	1.04
BDTI.control	1	1	1	1.04
BDTI.cxfir	1	1	1	1
BDTI.fft99	1	1.02	1.02	1.03
BDTI.iir	1	1	1	1
BDTI.lms	1	1.01	1.01	1.01
BDTI.ssfir	1	1.02	1.02	1.02
BDTI.vecmax	1	1.01	1.01	1.01
BDTI.vecprod	1	1.01	1.01	1.01
BDTI.vecsum	1	1.01	1.01	1.01
BDTI.viterbi	1	1.01	1.01	1.01
ITI.arrayaccess	1	1.03	1.1	1.1
ITI.bitaccess	1	1.02	1.02	1.02
ITI.case_test	1	1	1	1
ITI.ctrlstruct	1	1.03	1.03	1.03
ITI.fieldaccess	1	1	1	1
ITI.logop	1	1.01	1.01	1.01
ITI.param	1	1.07	1.09	1.09
ITI.polynome	1	1	1	1
ITI.recursive	1	1	1	1.01
ITI.squareroot	1	1	1	1
KERN.autcor	1	1	1	1
KERN.bitonic	1	1	1	1.19
KERN.bitrev	1	1.01	1.01	1.01
KERN.bsearch	1	1.01	1.01	1.01
KERN.copya	1	1	1	1
KERN.dct	1	1	1	1
KERN.dotprod	1	1	1	1
KERN.euclid	1	1.08	1.08	1.16
KERN.fir8	1	1.01	1.01	1
KERN.fircirc	1	1.01	1.01	1.01
KERN.floydallpairs	1	1.02	1.02	1.02
KERN.heapsort	1	1.06	0.99	1.18
KERN.kmpsearch	1	1.08	1.1	1.1
KERN.latanal	1	1.01	1.01	1.01
KERN.lsearch	1	1.06	1.06	1.06
KERN.max	1	1.02	1.02	1.02
KERN.maxindex	1	1	1	1
KERN.mergesort	1	1.05	1.05	1.08
KERN.quicksort	1	1.05	1.05	1.05
KERN.shellsort	1	1	1	1.08
KERN.strtrim	1	1	1	1
KERN.strwc	1	1	1	1
KERN.vadd	1	1	1	1
MUL.fir_int	1	1.01	1.01	1.01
MUL.jpeg	1	1.02	1.02	1.04
MUL.ucbqsort	1	1.03	1.03	1.03
STFD.stanford	1	1.01	1.01	1.01
G.Mean (50)	1	1.02	1.02	1.03

Table 4: Normalized performance in cycles for the KERNELS suite

Benchmark	Split	Edge motion		All
		w/o bl motion	w/ bl motion	
164.gzip	0.65	1	1	1.03
175.vpr	0.96	1	1.02	0.97
176.gcc	1.01	1	1.4	3.35
181.mcf	0.98	1	3.04	3.05
186.crafty	0.67	1	1.1	1.34
197.parser	0.91	1	1.43	2.23
253.perlbmk	0.88	1	1.08	1.09
255.vortex	0.99	1	1	1
256.bzip2	0.8	1	1.02	1.03
300.twolf	0.98	1	1.05	1.35
G.Mean (10)	0.87	1	1.23	1.47

Table 5: Reduction factor of the weight of moves

Table 5 gives the results of this experiment on Spec2000. On average, we divided the weight of move operations by a factor 1.23 and we observed no loss. For the mcf benchmark, we reduce this weight by a factor of 3. For the KERNELS, the block motion has nearly no effects when we run the same experiment. At the basic block scope, there are fewer opportunities for reduction of the size of parallel copies in these benchmarks compared to Spec2000. Indeed, we observed that the length of the basic blocks is generally smaller in these benchmarks and that there are fewer call sites (a call site puts additional constraints on coloring and thus favors parallel copy motion).

Performance impact Finally, we measured the performance impact of basic block motion in addition to the weight reduction of move operations. Table 3 (the two columns edge motion w/o and with block motion) shows the comparison in cycles on Spec2000 between the edge motion heuristic and the same heuristic followed by the block motion heuristic.

We see that this heuristic brings on the average an additional percent of performance compared to the edge motion. To be noted, we got a regression of one percent on the gzip benchmark. This regression is the result of a bad interaction between the block motion and the compiler post-scheduling phase. This is a limitation of the cost model implemented that does not account for the availability of resource slots. Thus, while in most cases the cost model is efficient, it may actually augment the schedule length, even when reducing the number of copies, due to a lack of resource at the point of insertion.

If we compare these results with the former split-everywhere strategy, we got an average speedup of 3%, with an improvement of 7% on mcf and 5% on parser. Again, we observed no performance loss. Considering the KERNELS, we had 3 improvements of 6%, 3% and 2% for respectively arrayaccess, param, and kmpsearch benchmarks. However, like for gzip in Spec2000 and for the same reason, we had one regression of 7% for the heapsort benchmark compared to edge motion only, and a regression of 1% compared to the split-everywhere strategy.

5.3 All together

To take advantage of the recoloring ability of motion inside basic blocks, we mentioned in Section 4.2 that we can integrate in the cost model of the local heuristic the optimized cost of placing a copy, not at bottom or top of a block, but also inside the block. In this section, we present the impact of this modelization on the overall performance.

Columns *All* in Table 3 and Table 4 report performances of respectively the Spec2000 and the KERNELS benchmarks. We have on the average 3% and 4% of improvements for respectively the KERNELS and the Spec2000 with no loss. This improves the previous edge motion plus block motion heuristic reported earlier at 2% and 3% improvement respectively.

We improve the performance of 5 over 6 benchmarks for Spec2000 and of 34 over 50 benchmarks for the KERNELS. We have 10 benchmarks with more than 5% of improvement in the KERNELS. In particular, 5 of these benchmarks are over 10% of improvement with greatest improvements for heapsort (18%) and bitonic (19%).

6. Conclusion

We introduced a new technique that we called parallel copy motion, which can be seen as a formalized tool for moving copies around in a control flow graph after register allocation has been performed. The goal is to reduce the global cost induced by the copies directly (additional instructions) or indirectly (edge splitting).

While our initial motivation was the motion of copies out of critical edges, this tool has been extended to the recoloring of arbitrary control-flow regions containing operations with register constraints. Thanks to the expansion of parallel copies into permutation of colors, the simple and sound theory on permutation motion, and the simple constraints on region boundaries, it is now easy to formalize a parallel copy motion problem including a cost model and with a freedom of motion from the granularity of an operation, to a basic block, and up to a complete region.

There are several possible applications to this technique. So far, we applied it for the problem of getting out of a colored SSA code as provided by a decoupled register allocation algorithm over SSA.

For this out-of-colored-SSA problem, we used the parallel copy motion technique as an enabler for moving away from critical edges the copies introduced by ϕ-functions, when it is profitable, or simply when the edge cannot be split, as it is the case for abnormal edges present in compiler code generators for C and C++. We have indicated that the permutation motion can be blocked in the presence of *multiplexing regions* where all critical edges are abnormal. In this case, we propose to use classical graph coloring techniques in order to recolor the multiplexing regions, however possibly with additional spills. Nevertheless, in practice, the compiler hardly generates such regions (actually never in our experiments), thus it does not appear to be an issue for performance.

In the context of our out-of-colored-SSA problem, for the multiple-issues VLIW architecture for which we are compiling, we got significant performance improvements (4% average speedup for the C integer subset of Spec2000 and 3% for the KERNELS) compared to the edge-splitting approach generally used. More generally, we have shown that not only critical edge splitting can be completely avoided when necessary, but also that one can benefit from having a cost model to drive the edge splitting decision. In our context, we got a reduction of the number of split critical edges by a factor of two when using a cost model, which demonstrates that edge splitting actually pays-off only once over two on average.

We believe that discovering that parallel copies can be easily moved is a major breakthrough for out-of-SSA translations. Up to now, it was in general considered that placing copies on edges would require to split them, which is not necessarily the best approach. For this reason, people tried to introduce copies directly at the borders of basic blocks since the discovery of SSA, starting with the algorithm in [6] up to the out-of-SSA translation in [17] and [3]. Recently, the idea of doing register allocation while still under SSA was developed. The goal is to use the nice properties of SSA for a longer time and, amongst them, the fact that the interference graph is chordal, hence easy to color. However, the drawback is that going out of SSA introduces parallel copies on edges. A recoloring technique was proposed in [11] to coalesce the copies on these edges, but splitting edges is still necessary whenever the coalescing fails. Last but not least, register allocators used for JIT com-

pilation, mostly variants of linear scans, perform poor coalescing and could benefit from a fast parallel copy motion post-phase.

Bibliography

[1] A. W. Appel and L. George. Optimal spilling for CISC machines with few registers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*, pages 243–253. ACM Press, 2001.

[2] T. Ball and J. R. Larus. Branch prediction for free. *SIGPLAN Notices*, 28(6):300–313, 1993.

[3] B. Boissinot, A. Darte, B. Dupont de Dinechin, C. Guillon, and F. Rastello. Revisiting out-of-ssa translation for correctness, code quality, and efficiency. In *International Symposium on Code Generation and Optimization (CGO'09)*. IEEE Computer Society Press, Mar. 2009.

[4] F. Bouchez, A. Darte, C. Guillon, and F. Rastello. Register allocation: What does the NP-completeness proof of Chaitin et al. really prove? or revisiting register allocation: Why and how. In *19th International Workshop on Languages and Compilers for Parallel Computing (LCPC'06)*, New Orleans, USA, Nov. 2006.

[5] P. Brisk, F. Dabiri, J. Macbeth, and M. Sarrafzadeh. Polynomial time graph coloring register allocation. In *14th International Workshop on Logic and Synthesis*, June 2005.

[6] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[7] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, 1981.

[8] S. M. Freudenberger, T. R. Gross, and P. G. Lowney. Avoidance and suppression of compensation code in a trace scheduling compiler. *ACM Transactions on Programming Languages and Systems*, 16(4): 1156–1214, 1994.

[9] L. George and A. W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.

[10] S. Hack. *Register Allocation for Programs in SSA Form*. PhD thesis, Universität Karlsruhe, Oct. 2007.

[11] S. Hack and G. Goos. Copy coalescing by graph recoloring. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 227–237, New York, NY, USA, 2008. ACM.

[12] S. Hack, D. Grund, and G. Goos. Register allocation for programs in SSA-form. In *International Conference on Compiler Construction (CC'06)*, volume 3923 of *LNCS*. Springer, 2006.

[13] F. M. Q. Pereira and J. Palsberg. SSA elimination after register allocation. In *18th International Conference on Compiler Construction (CC'09)*, volume 5501 of *LNCS*, pages 158–173, York, UK, Mar. 2009. Springer.

[14] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

[15] L. Rideau, B. P. Serpette, and X. Leroy. Tilting at windmills with Coq: Formal verification of a compilation algorithm for parallel moves. *Journal of Automated Reasoning*, 40(4):307–326, 2008.

[16] V. Sarkar and R. Barik. Extended linear scan: An alternate foundation for global register allocation. In *International Conference on Compiler Construction (CC'07)*, volume 4420 of *LNCS*, pages 141–155. Springer, 2007.

[17] V. C. Sreedhar, R. D. Ju, D. M. Gillies, and V. Santhanam. Translating out of static single assignment form. In A. Cortesi and G. Filé, editors, *6th International Symposium on Static Analysis*, volume 1694 of *LNCS*, pages 194–210. Springer, 1999.

[18] O. Traub, G. H. Holloway, and M. D. Smith. Quality and speed in linear-scan register allocation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 142–151. ACM Press, 1998.

[19] C. Wimmer and H. Mössenböck. Optimized interval splitting in a linear scan register allocator. In *1st International Conference on Virtual Execution Environments (VEE)*, 2005.